並列プログラムからのハードウェア自動生成システムの検討

松井　誠二

内容梗概

本論文では、並列プログラミング言語の１つである OpenMP からハードウェア記述言語である VHDL を生成するシステムの処理方式を検討する。OpenMP とは、共有メモリ並列プログラミング言語であり、逐次プログラムに並列化指示文を挿入することにより、比較的簡単に並列プログラムを作成することができる。本システムでは、M コンパイラと T コンパイラの２つのコンパイラで構成されている。本研究で検討する M コンパイラは、並列化指示文を解析する並列構文解析部、並列リージョンのハードウェア処理を実現する並列リージョン処理部、メモリアクセスユニットや同期ユニットを生成する同期処理部により、OpenMP を C プログラムに変換する。さらに、変換された C プログラムを T コンパイラに通すことにより、各 PE およびユニットを VHDL 記述で生成する。VHDL を市販の VHDL コンパイラに通すことにより、FPGA 上に実装できるハードウェア構成データに変換できる。

本研究では、並列構文、ワークシェアリング構文、データスコープ属性等の解析方法と並列リージョンを実行する処理方法、同期や相互排除に関する解析方法について検討した。また、共有メモリの実現方法が重要であるので、各種並列ハードウェアの調査を行い、本システムでの同期やメモリアクセスの実現方法に関する並列ハードウェア構成例を示した。

目次

図目次

表目次

１．はじめに

近年、半導体集積技術での速度向上及びパイプライン処理やスーパースカラなどの並列処理により、シングルプロセッサの性能は、飛躍的に向上してきている。しかし、クロックアップによる性能向上は、現在の１０倍前後が限界と見られており、効果的なパイプライン処理にいたっても、１０～１５段が限界と言われている。よって、シングルプロセッサによる性能向上は近い将来限界がくると考えられている[18]。

その一方、マルチプロセッシングによる並列処理に基づき、高速処理を実現する方法が有望視されている。以前に比べ非常に安価になった PC を複数台、高速ネットワークでつなぐことによりマルチプロセッシングを実現する PC クラスタの登場や複数プロセッサを１チップに実装する、オンチップマルチプロセッサシステムの実現が近く実現すると思われる。これらマルチプロセッサによる性能向上は今後必須になってくると考えられる。

このマルチプロセッサの能力を十分に引き出すためには、並列プログラミングが欠かせなくなる。この並列プログラミングには PVM に代表される分散メモリ型並列プログラミングと OpenMP に代表される共有メモリ型並列プログラミングがある[13]。近年では、これまで主流だった分散メモリ型から共有メモリ型に移行しつつあり、そのため本研究ではターゲットプログラミング言語を OpenMP とした。この OpenMP の特徴は、逐次プログラムに OprnMP ライブラリによって用意された並列化指示文を挿入することにより、容易に並列プログラミングができるという利点がある。

また、大規模な問題を解決する方法として、並列処理による高速化の他に、ハードウェアによる高速化が挙げられる。ある問題に特化したハードウェアを生成することができれば、実行時間の大幅な短縮が見込め、汎用的な並列マシンを用意するよりも遥かにコンパクトで低コストになる。ハードウェアを自在に構成できる FPGA(Field Programmable Gate Array)が登場し、ハードウェア開発が短期間、低コストで行えるようになることから先に述べた事柄が可能となる。

システム設計手法として、ハードウェア/ソフトウェア・コデザイン（以下 H/S コデザイン）がある[28][29]。H/S コデザインは、大規模かつ複雑なシステムをハードウェア及びソフトウェアのトレードオフを考慮して、最適設計を行う手法のことである。H/S コデザインは、従来、熟練者の勘と経験によって行われてきたが、システムの複雑化に伴って、グループを編成して、分担設計するようになった。さらに、CAD や FPGA、HDL(Hardware Description Language)の登場により、ハードウェアとソフトウェアの協調関係を探りながらシステム設計できる環境が整ってきたこともあって、H/S コデザインの研究対象はシステム単体の設計からシステムの自動生成へと変遷していった。そこで、本研究室では以前、逐次 C プログラムからのハードウェア自動生成システムを構築しており、KITE マイクロプロセッサと比べて、実行速度が２倍前後の成果があがっている。

これらの研究背景から、共有メモリ型並列プログラミング言語 OpenMP からコンパクトな専用ハードウェアを生成するシステムの構築を検討する。また、同期処理の検討、共有メモリの実現方法、及び FPGA 上での実現方法についても検討する。共有メモリの実現方法には、一般的に２つの方法が挙げられる。１つは SMP と呼ばれる対称型マルチプロセッサに用いられる共有メモリで、物理的に１ヵ所にグローバルメモリとして集中配置するメモリ構成である。各プロセッサと１つの共有メモリとが共有バスで結合されるため、そこに流れるメモリトランザクションを管理しやすく、キャッシュ自身が能動的にキャッシュコヒーレンスを維持しやすい。この方式をスヌープキャッシュ方式と呼ばれる。しかし、対象問題が大規模になれば、すべてのメモリトランザクションを管理するのは困難で非効率である。その場合、もう１つの分散共有メモリ構成を用いるのがよい。プロセッサ単位でローカルメモリを分散配置しており、それらを相互結合網で結合する。その際、キャッシュとは別にキャッシュコヒーレンスを維持するためのコントローラがシステム内に存在し、各キャッシュはその指示に従って受動的に自分自身のコヒーレンスを維持する。この方式をディレクトリ方式と呼ぶ。こうすることにより、対象問題の大規模化に対しての対応できると考える。

本論文では、まず、第２章で共有メモリ型並列プログラミングモデルである OpenMP に関して述べ、第３章で、OpenMP からのハードウェア自動生成システムに関して述べる。第４章では、並列ハードウェアの調査や本システムで生成されるハードウェアの同期やメモリアクセスの方法及び並列ハードウェア構成例について述べる。最後に第５章では、本論文の結言と今後の課題を示す

2. 共有メモリ型並列プログラミング言語 OpenMP

近年、並列プログラミング環境が変化しつつある。以前までの分散メモリ型プログラミングでは、並列処理に通信処理（メッセージパッシングなど）が不可欠であり、相当なオーバーヘッドが見込まれていた。また、プログラミング自体も非常に複雑となり、ある程度の知識レベルでないと困難であった。しかし、対称マルチプロセッサ（SMP）のような共有メモリ型マシンや CC-NUMA 及びソフトウェア DSM のような分散共有メモリ型マシンの普及により共有メモリ型のプログラミングが注目されてきている。OpenMP とは、その共有メモリ型プログラミング言語の１つで、今後最も注目される言語であると考えられている。OpenMP の特徴として、逐次プログラムに並列化指示文を挿入することにより、容易に並列プログラミングが可能となることが挙げられる。その例を図１に示す。OpenMP のベース言語は C/C++/Fortran の３種類あるが、ここでは C 言語ベースのプログラム例を示す。

```
#include <stdio.h>

int i,n=100,sum;

void main()
{
 #pragma omp for reduction(+:sum) private(i)
   for(i =1 ; i = n; i++)
     sum = sum + i;

   return = 0;
}
```

図 1　プログラム例

図１で示した通り、逐次 C プログラムに#pragma omp で始まる並列化指示文を挿入することにより、並列化している。この例では、並列化指示文以下の for 文をデータ並列化している。すなわち、１～N の和を OpenMP 環境変数もしくは OpenMP 関数により与えられるスレッド数によって分割し、並列実行している。図２に OpenMP の一般的なアーキテクチャを示す。

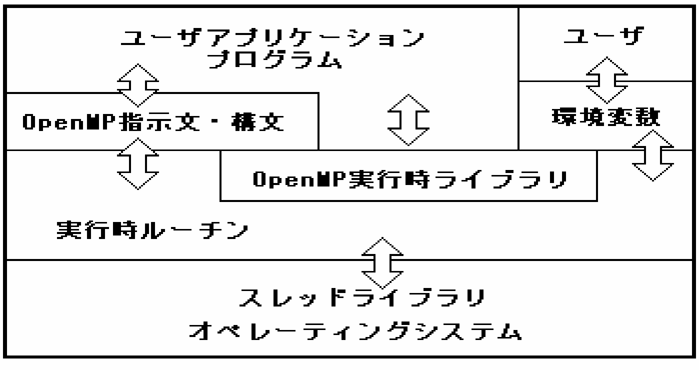


図 2　 OpenMP アーキテクチャ

2.1 並列リージョン

OpenMP の概念として最も重要なものに並列リージョンがある。図３にその概念図を示す。OpenMP は、fork-join 型の並列実行モデルを採用している。

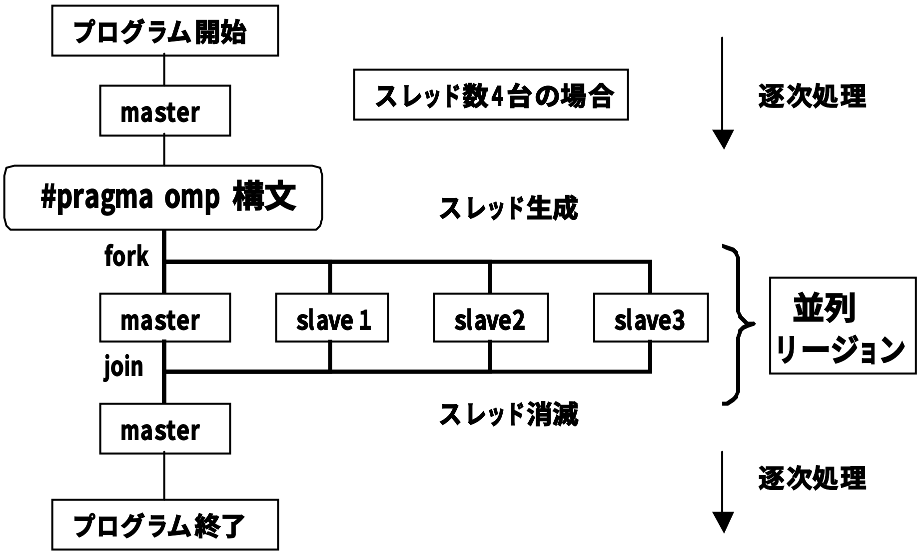


図 3 並列リージョン

プログラムの開始から並列化指示文（#pragma omp 構文）まではマスタスレッドにより逐次処理を行う。並列化指示文が現れてから、スレッドライブラリオペレーティングシステムはチームと呼ばれる複数のスレーブスレッドを生成し、実行時ルーチンとしてマスタスレッド及び複数のスレーブスレッドに fork する。並列実行を行うこの領域を並列リージョンと呼ぶ。並列リージョンが終わり次第、スレッドライブラリオペレーティングシステムはスレーブスレッドを消滅させ、マスタスレッドに join し、引き続き逐次処理を行う。また、join 時には暗黙のバリア同期が取られる。

並列リージョン内の並列技法はプログラマに依存している。つまり、OpenMP での並列効果はプログラマの技量によるところが大きい。例えば、データ並列は自動ブロック分割により実現され、並列効果は一様に見込めるが、タスク並列では、並列リージョン内のどのスレッドにどのタスクを実行させるかの指定は一切行えないので、あらかじめ考慮しないと期待する並列効果は得られない場合がある。

2.2 並列化指示文

OpenMP からハードウェアを自動生成するにあたって、OpenMP 独特の並列化指示文を解析する必要がある。並列化指示文には大きく分けて３種類ある。以下に示す。

2.2.1　並列構文

並列構文とは、並列実行を開始する基本的な構文で、並列リージョンを定義する。C/C++ベースでの並列構文の表現は次のようになる。

```
#pragma omp parallel [clause]...
{
    並列リージョン
}
```

この構文で示された範囲において fork-join され、並列リージョンとして並列実行される。生成されたスレッドには、自動的にスレッド番号が与えられ、元の master スレッドはスレッド番号が０となる。チームが一旦生成されると、チームのスレッド数を変更することはできない。このため、並列リージョン実行中にスレッド数が変わることはない。並列リージョンの最後では、暗黙にバリア同期が実行される。並列リージョンの終了後、チームの master スレッドのみが逐次実行する。

2.2.2　ワークシェアリング構文

ワークシェアリング構文は、この構文に到達したチームのメンバに、対応するステートメントを分割して実行させる。このとき、新たにスレッドは生成されない。OpenMP では、次のワークシェアリング構文を定義している。

表 1　ワークシェアリング構文

ワークシェアリング構文	C/C++での表現	効果
for 構文	#pragma omp for [clause]	データ並列
sections 構文	#pragma omp sections [clause]	タスク並列
single 構文	#pragma omp single [clause]	逐次処理

（１） for 構文

for 構文は、対応するループのイタレーション処理を並列で実行すべきリージョンとして定義する。図４に概念図を示す。

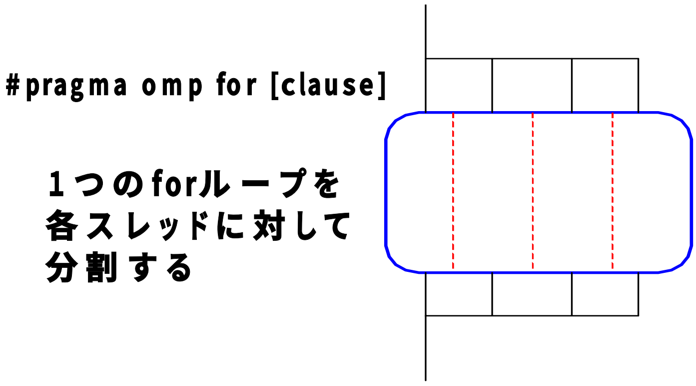


図 4　for 構文

for ループのイタレーションはすでに存在するスレッド間でブロック分割される。for 構文の最後では暗黙のバリア同期が実行される。

（2） sections 構文

sections 構文は、チーム内のスレッドで分割して実行する構文の集合を定義する、非繰り返しのワークシェアリング構文である。概念図を図５に示す。

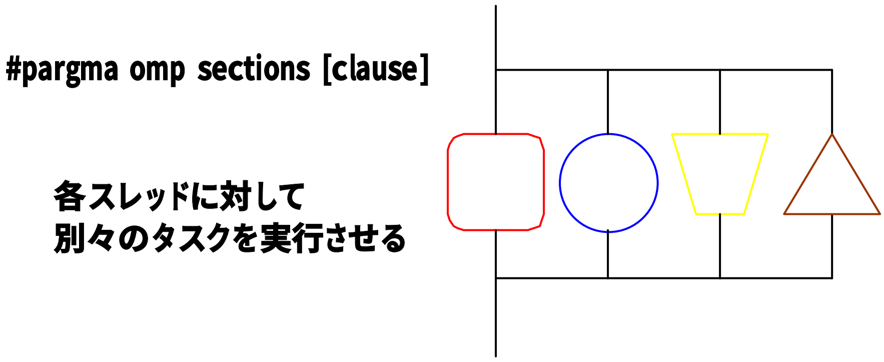


図 5　sections 構文

各セクションはチーム内のスレッドにより１度だけ実行される。Sections 構文の最後では暗黙のバリア同期が実行される。

（3） single 構文

single 構文は、対応する構造ブロックがチーム内の１つのスレッド（必ずしも master スレッドでなくてもよい）のみで実行されることを指示する構文である。single 構文の最後では暗黙のバリア同期が実行される。

2.2.3　同期構文

同期構文とは、大きく分けて２つの同期を指示する構文である。表２に示す。

表 2　同期構文

排他的制御に関する同期	critical 構文	#pragma omp critical	１スレッドのみアクセス可
	atomic 構文	#pragma omp atomic	critical と同様。以下の１文にのみ有効
	実行時 library	omp_set_lock() omp_set_unlock()など	様々な lock 関数
イベントに関する同期	barrier 構文	#pragma omp barrier	強制的にバリア同期を実行
	ordered 構文	#pragma omp ordered	逐次ループの順序で実行
	master 構文	#pragma omp master	マスタスレッドで実行

（１）　排他的制御に関する同期

主にデータレース（アクセス競合）を防ぐための同期である。OpenMP が共有メモリを対象としているため、複数のスレッドが局所変数を持つ場合、その変数に対するデータレースが発生する。（２．３参照）メモリアクセスの制御によって解決しなければならない場合、これらの同期構文を用いる。

（２）　イベントに関する同期

並列構文やワークシェアリング構文の最後では、暗黙のバリア同期が実行される。しかし、並列リージョン内において別の場所で同期を取りたい場合、これらの同期構文を用いて同期を取る必要がある。

2.3 データスコープ属性指示文

いくつかの指示節では、ユーザが指示節(clause)を用いて、そのリージョンの実行中に変数のスコープ属性を制御することができる。スコープ属性指示節は、その指示節が指定されている指示文の文脈有効範囲にある変数にのみ適用される。表３に示す。

表 3　データスコープ属性指示節

属性名	説明
private	変数を局所化する。
shared	変数をスレッド間で共有化する。default では shared。
reduction	変数を演算子でリダクションする。
firstprivate	private と同様であるが、直前の値で初期化される。
lastprivate	private と同様。構文終了時に逐次実行された場合の最終値を反映する。

主に変数の局所化、共有化及び、reduction 演算の３つが重要である。各スレッドに対して変数を局所化するのか、共有化するのかは極めて重要である。安易なデータスコープ属性指定は date race を引き起こし、期待した演算結果および並列効果は得られない。また、reduction で指定できる演算子には、（+,－,＊,&,^,|,&&,||）があり、指定された並列リージョンの最後で演算される。

3. OpenMP からのハードウェア自動生成システム

本研究では、並列プログラムからのハードウェア自動生成システムを検討するにあたり、ターゲットプログラミング言語として、OpenMP プログラムを前提としている。また、出力される HDL（ハードウェア記述言語）として、VHDL を想定している。次に、システムの具体的な構成を示す。

3.1 システム構成

図６にシステム構成を示す。

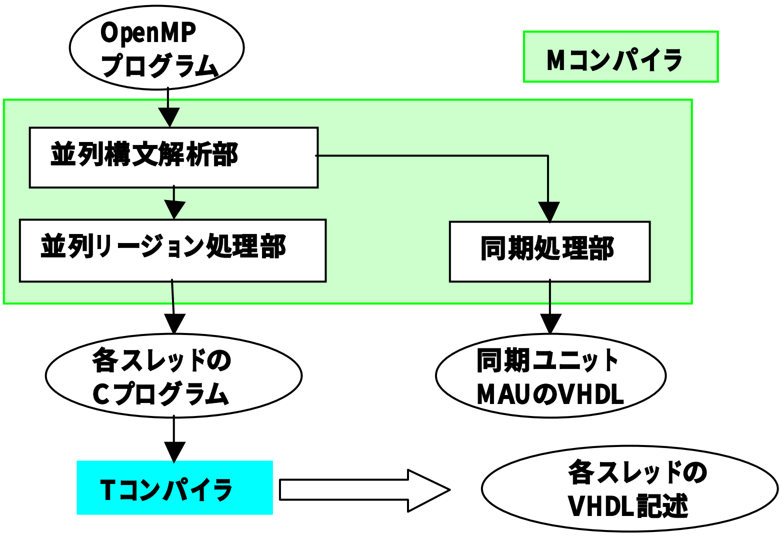


図 6　システム構成図

本システムはまず C ベースの OpenMP プログラムを入力とし、OpenMP 独特の並列化指示文を並列構文解析部により解析する。ここでは、スレッドごとの処理範囲に分割する処理、プライベート変数のコピー処理などを行う。その後、各スレッドに対応する処理を C プログラムに変換する並列リージョン処理部に通す。変換された C プログラムは、次に T コンパイラに運ばれる。ここで T コンパイラとは、逐次 C プログラムから VHDL 記述を自動生成するコンパイラである[16]。また同時に、同期処理部により OpenMP 関数から抽出した並列情報を基にメモリアクセスや同期処理をハードウェアで実現するためのユニットを VHDL 記述で生成する。

最後に、本システムにより生成された VHDL 記述から機能シミュレーションを行う。また論理合成・配置配線ツールを用いて、FPGA 上にロードするためのハードウェア構成データを得て、FPGA ボード上で動作検証をする。

3.2　C 言語からのハードウェア自動生成システム

M コンパイラから出力された各スレッドに対する C プログラムは、T コンパイラによって VHDL 記述に変換される。つまり、T コンパイラとは C 言語からのハードウェア自動生成システムである。図 7 に T コンパイラシステム構成図をしめす。

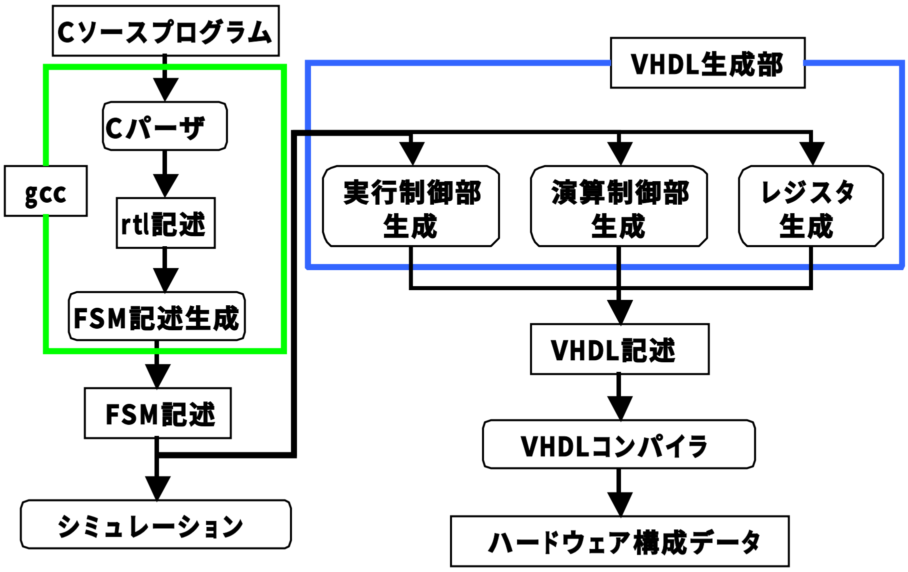


図 7　T コンパイラシステム構成

T コンパイラは C 言語パーザとして、gcc を利用しているため、基本的には C 言語構文のすべてを受け付ける。大きな特徴として、ループ構造（可変ループ回数も含む）、再起呼び出し、配列等の構文もサポートしている。C プログラムからハードウェアに変換する方法としては、C プログラムを一旦有限ステートマシン（FSM）に変換し、そのステートマシンをハードウェア化するという手法をとっている。このステートマシンをステートマシン記述（FSML）として出力する。出力された FSML を VHDL 生成部により、必要なレジスタ数や機能を割り出し、不必要な論理回路は極力削除することで、必要最小限のハードウェアの生成を行う。最後に、生成された VHDL 記述は市販の VHDL コンパイラによりハードウェア構成データに変換できる。このハードウェア構成データは、FPGA に実装できるデータである。

3.3 並列構文解析部

まず、OpenMP プログラムは、並列構文解析部を通る。この並列構文解析部とは、OpenMP 独特の並列化指示文を解析し、その結果を次の並列リージョン処理部及び同期処理部に反映させる。また、スレッド数の所得やスレッド ID の発行も行う。

図 8 に並列構文解析部の処理を示す。

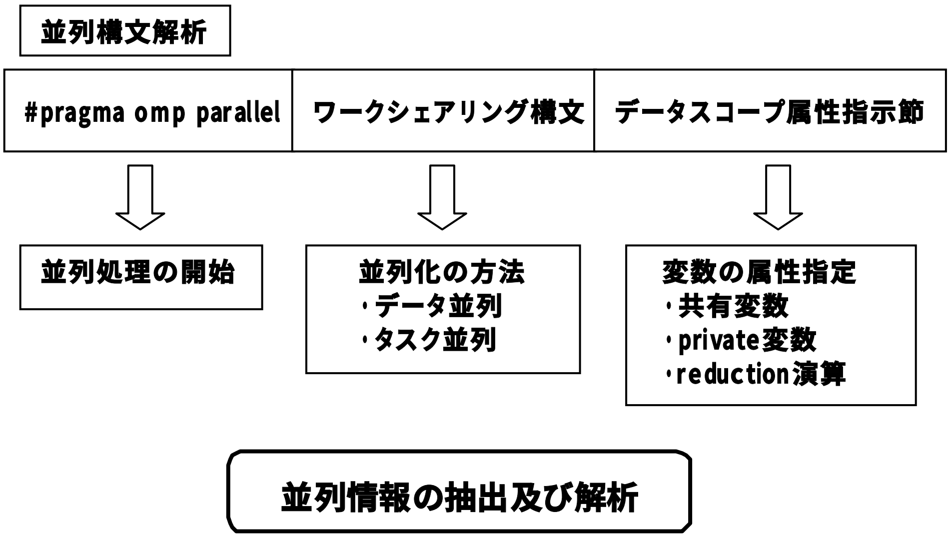


図 8　並列構文解析部の処理

OpenMP の並列化指示文の構成は主に３つに分かれている。並列構文、ワークシェアリング構文、データスコープ属性指示節である。この３要素に対してそれぞれ解析する必要がある。

3.3.1　並列構文解析

並列構文解析での主な処理を図９に示す。

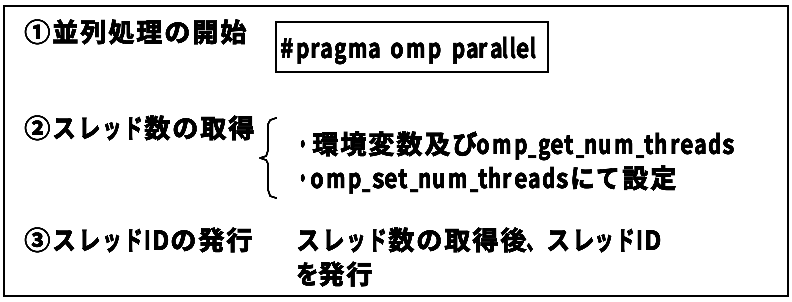


図 9　並列構文解析

OpenMP の実行時処理では、並列化指示文"#pragma omp parallel～"から並列リージョンが開始され、その時点で設定されているスレッド数に基づいたスレッドが生成される。本システムでは、この並列化指示文が現れた時点で、その有効範囲を並列リージョンであると解釈し、その時点で設定されているスレッド数を取得する。設定の方法は環境変数（実行時にあらかじめ与えておく）で行う方法とプログラム中で OpenMP 関数の１つである omp_set_num_threads( )を用いて設定する方法の２つある。設定されたスレッド数の個数分だけ並列リージョン中のプログラムをコピーする。また、データ

並列では、各スレッドの処理範囲を算出する必要があり、その際用いるスレッド ID も取得しておく。スレッド ID とは、各スレッドに与えるスレッド番号であり、マスタスレッドをスレッド ID＝１とする。

3.3.2　ワークシェアリング構文解析

並列化指示文中において、チーム内のスレッドで分担して実行する部分を指定できる構文にワークシェアリング構文がある。ワークシェアリング構文解析の主な処理を図 10 に示す。

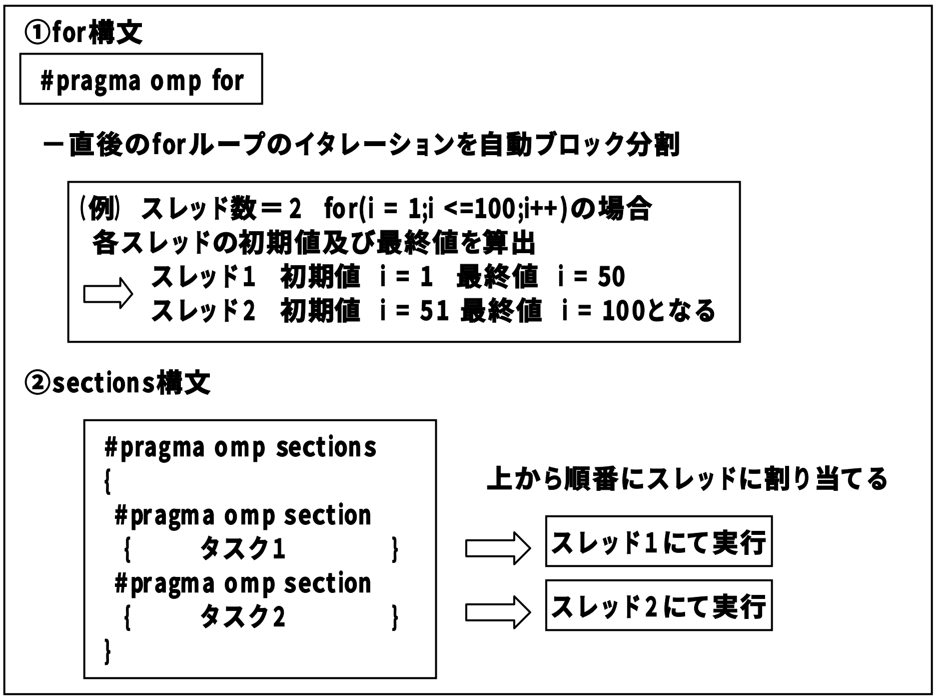


図 10　ワークシェアリング構文解析

本システムで取り上げるワークシェアリング構文には主に２つの種類がある。１つは、データ並列を実行するときに用いられ、for ループのイタレーション（繰り返し）をブロック分割により分担して実行する for 構文がある。この時、プログラマがブロック分割を行うのではなく、自動的に分割する。つまり、各スレッドに対する for ループの初期値及び最終値を算出する必要がある。その算出方法は次の式の通りである。

スレッド T_NUM の初期値 = (master の最終値/N)＊T_NUM+１
スレッド T_NUM の最終値 = (master の最終値/N)＊(T_NUM+１)

ここで、N はスレッド数、T_NUM はスレッド ID である。

また、もう１つは、タスク並列を実行するときに用いられ、独立したタスクをセクシ

ョンごとに分担してそれぞれのスレッドに割り当てる sections 構文がある。この構文では、各スレッドに担当するタスクを割り当てるために、自身の担当ではないタスク部分のプログラムはコピーしないことで、タスク並列を実行する。

3.3.3　データスコープ属性指示節解析

共有メモリ型プログラミングモデルである OpenMP では、変数の属性（データスコープ属性）が非常に重要になってくる。データスコープ属性指示節解析で行う主な処理を図１１で示す。

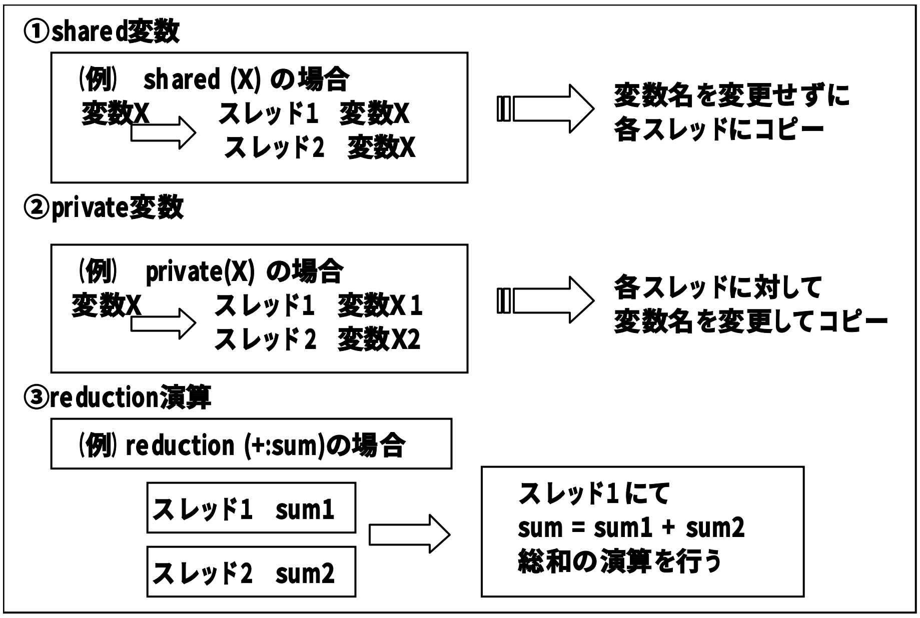


図 11　データスコープ属性指示節解析

本システムで取り上げるデータスコープ属性指示節は３種類である。１つ目は、shared 変数（共有変数）の指示である。共有変数であるので各スレッドにプログラムをコピーする際、変数名は変更する必要がない。つまり、各スレッドは同一変数にアクセスし、MAU（メモリアクセスユニット）によって、アクセススケジュールは管理される。２つ目は、変数を各スレッドで局所化したい場合に用いる private 変数の指示である。この場合は、プログラムのコピー時に、変数名を変更してコピーを行うことで、まったく別の変数として扱え、他のスレッドからアクセスされることがなくなる。最後に、特殊な演算方法である reduction 演算もサポートしたい。この reduction 演算で指定する変数は必ず private 変数として扱われ、構文の最後で指定した演算を行う。本システムでは、演算を行うスレッドをマスタスレッドに限定する。よって、マスタスレッドのプログラムには、reduction 演算を行うプログラムが自動的に追加される。

3.4 並列リージョン処理部

並列リージョン処理部では、並列構文解析部で行った解析結果に基づいて、元プログラムから各スレッド用のプログラムをコピーする。並列構文解析及びデータスコープ属性指示節解析に関しては、どのプログラムにおいても処理は異ならないが、ワークシェアリング構文解析の結果如何では、並列処理が大きく異なってくる。前述した通り、ワークシェアリング構文により指示できる並列処理には、データ並列とタスク並列の２種類ある。次にそれぞれの処理方式を示していく。

3.4.1　データ分割

図１２に１から 100 の和でのデータ並列を行ったときの処理方式を示す。

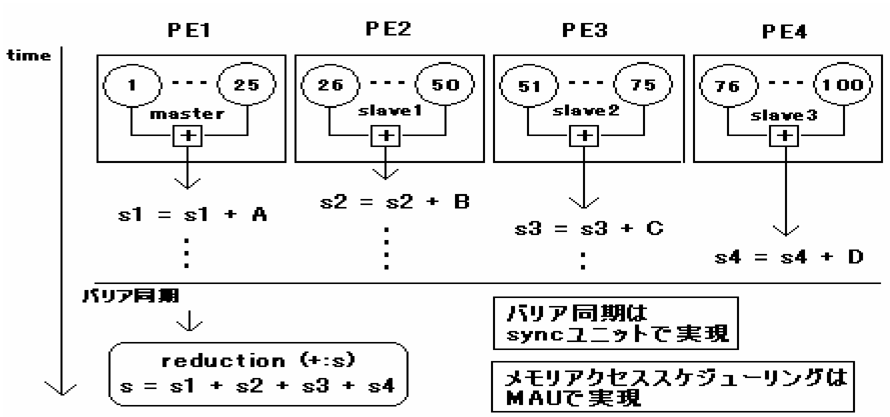


図 12　データ並列の処理方式

各スレッドが解析結果による処理範囲を計算し、処理の最後でバリア同期を取っている。その後、reduction 演算で総和を求めている。private 変数は別々の変数として扱う。ここで、バリア同期は同期ユニットで実現され、共有メモリのメモリアクセススケジューリングは MAU（メモリアクセスユニット）で実現する。

3.4.2　タスク分割

図１３にタスク並列を行ったときの処理方式を示す。

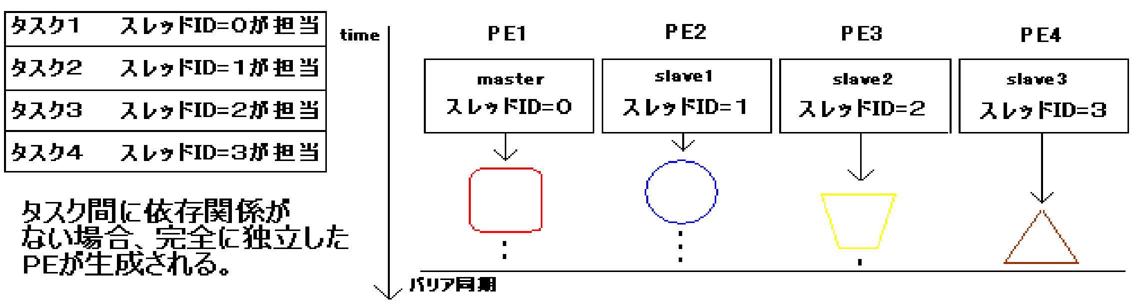


図 13　タスク並列の処理方式

タスク並列ではタスク間の依存がない時、各スレッドは完全に独立した PE として生成される。それぞれのタスクが終了した時点で、データ並列と同様にバリア同期が実現される。

3.5 同期処理部

同期処理部での処理を図１４に示す。

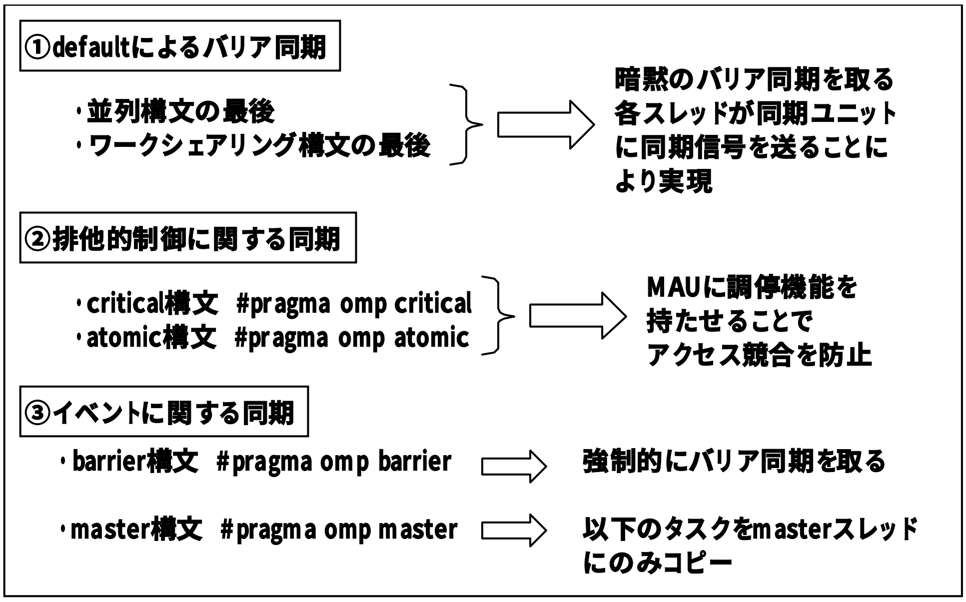


図 14　同期処理部の処理

default によるバリア同期とイベント同期構文である barrier 構文の処理に関しては、同期ユニットが行う。各スレッドに同期信号を付加し、バリア同期が必要なところで同期ユニットにその信号を送信する。同期ユニットでは、処理不許可信号を各スレッドに送信し、すべての同期信号を受信すると、処理許可信号をすべてのスレッドに送信する。

また、排他的制御に関する同期は、主にアクセス競合を防止するための同期であるため、各スレッドにアクセス信号を付加し、メモリにアクセスしたいスレッドは、まず MAU にアクセス信号を送信し、MAU はそのスレッドに対してアクセス許可信号を送信する。同時に、その他のスレッドにはアクセス不許可信号を送信する。こうすることで、アクセス競合を防止することができる。

4. 並列ハードウェアの検討

4.1 並列ハードウェアの調査

本システムで生成される共有メモリ型並列ハードウェアのアーキテクチャを考察する。共有メモリ型並列ハードウェアのアーキテクチャを決定する最も大きな要因は、プロセッサ間でデータ授受をどのように行うか、および、メモリをどのように構成するかである。まず、共有メモリを物理的に１ヵ所にグローバルメモリとして集中配置する対称型マルチプロセッサ（以下 SMP）がある。これは、小規模な並列計算機では容易に構成可能であり、１９８０年代前半に商用化された並列計算機にはこの方式を採用した共有バス型マルチプロセッサが多い。また、プロセッサごとにローカルメモリとして分散配置する分散共有メモリ方式がある。これは、大規模なシステムだとハードウェア構成が容易となる利点がある。この２つの方式を図１５に示す。

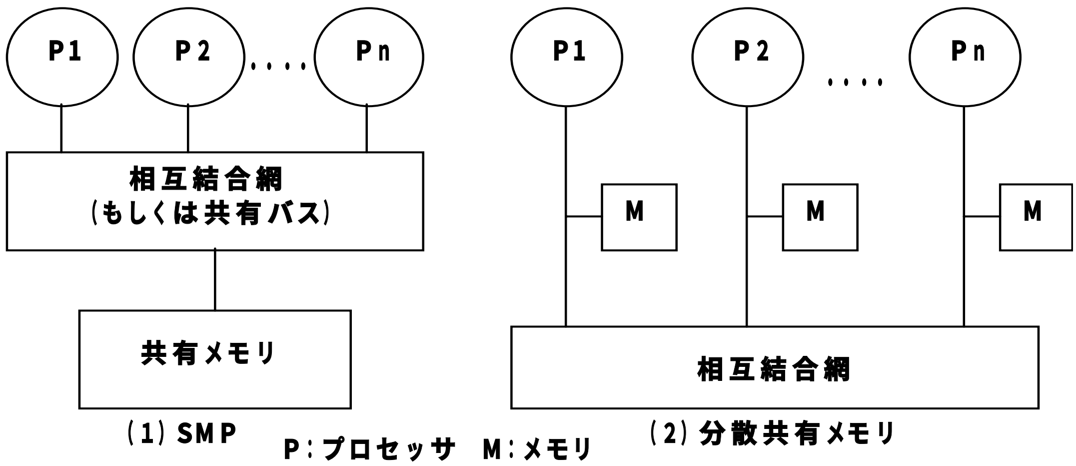


図 15　SMP 及び分散共有メモリ

さらに分散共有メモリの場合、さらにメモリアドレスに対するアクセスレイテンシが一定か否かで２つに分かれる。１つは、UMA（uniform memory access）モデルであり、すべてのプロセッサから同一時間で共有メモリにアクセス可能なモデルである。もう１つは、NUMA（non-uniform memory access）モデルであり、各プロセッサから見た共有メモリの距離はアドレスにより遠近が生じアクセス時間も異なる。ここで、データ配置の点では UMA に利点がある。NUMA では、データ分割/配置の際にデータ参照の局所性を考慮に入れる必要がある。

SMP か分散共有メモリか、および UMA か NUMA かを問わず、プロセッサが共有メモリ内のデータ（共有データと呼ぶ）へのアクセスの度に共有メモリに直接アクセスすると、相互結合網のトラフィックが増大してしまい、かつ、メモリアクセス時間が増加する。解決策として、共有データをプロセッサにプライベートなキャッシュへキャッシングすることが挙げられる。しかし、共有データのプライベートキャッシュへのキャ

ッシングを許すと、１個の共有データのコピーが複数のキャッシュに同時に存在する可能性が生じる。このとき、あるプロセッサが自キャッシュ内のある共有データのコピーに書き込みを行うと、他キャッシュ内に存在する当該共有データの別のコピーとの間で値が一致しなくなる可能性がある。これは、キャッシュコヒーレンス問題であり、その概念を図１６に示す。

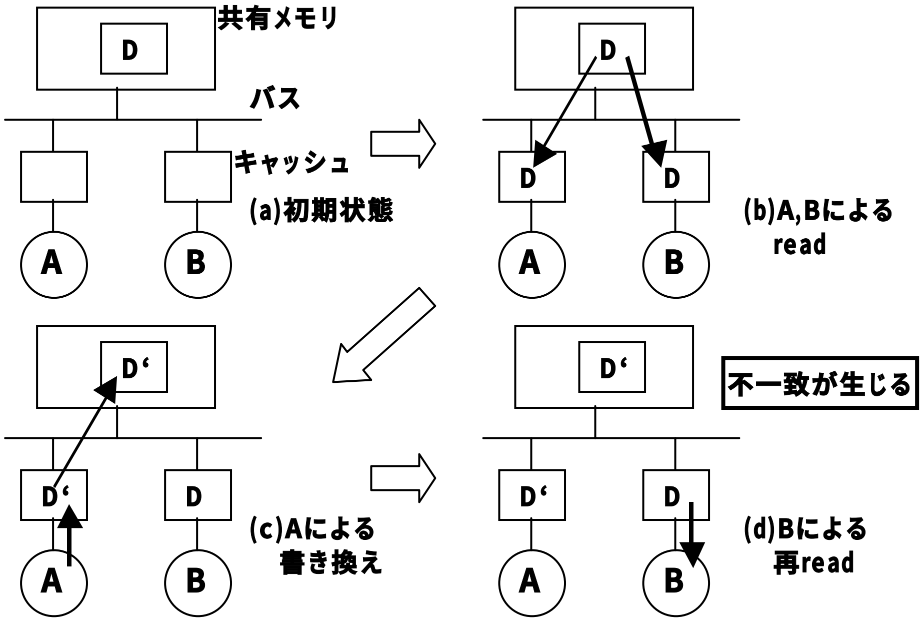


図 16　キャッシュコヒーレンス問題

キャッシュコヒーレンス問題のハードウェアによる解決策には多くのものが提案されているが、これらは主に以下の観点から分類する。

（1） スヌープキャッシュ法

キャッシュ自身が能動的にコヒーレンスを維持する。すなわち、各キャッシュは相互結合網上を流れるすべてのメモリトランザクションを監視しており、自分自身に影響を及ぼすトランザクションを検出するとコヒーレンスを維持するために適切な処理を取る。キャッシュディレクトリは必然的に各キャッシュに分配配置される。

（2） ディレクトリ法

キャッシュとは別にコヒーレンス維持を司るコントロールがシステム内に存在し、各キャッシュはその指示に従って受動的にコヒーレンスを維持する。コントローラがどのデータをキャッシングしているかという情報をディレクトリとして管理する。

また、共有メモリとキャッシュメモリという２レベルのメモリ階層を構成であれば CC－NUMA(cache-coherent NUMA)モデルと呼ばれ、すべてのメモリが概念上キャッシュメモリの１レベルだけであれば COMA(cache only memory architecture)モデルと呼ばれる。

4.2 同期とメモリアクセス

本システムで自動生成される並列ハードウェアでもっとも重要かつ困難なのが、同期及びメモリアクセスの問題である。そこで考察するハードウェアアーキテクチャを２つに分類して、同期とメモリアクセスについて述べていく。前節で調査したことから、ハードウェアの分類を表５に示す。

表 4　ハードウェアの分類

	ハードウェア例①	ハードウェア例②
対象問題の規模	小規模	大規模
メモリアーキテクチャ	SMP	分散共有メモリ方式
ネットワーク	共有バス	相互結合網
キャッシュコヒーレンス	スヌープキャッシュ方式	ディレクトリ方式
メモリアクセス方式	調停式 MAU	CC-NUMA 式 MAU
FPGA	1FPGA－複数スレッド	１FPGA－１スレッド

（１）小規模並列ハードウェアの例

対象とする問題が比較的小規模な場合、ハードウェア量の削減などを考慮に入れるため、１FPGA に複数のスレッドを実装する。それらに対して共有バスを用いて、共有メモリの役割を果たす外部 RAM と結合し SMP を構成する。このとき、共有バス型マルチプロセッサに広く用いられているスヌープキャッシュ方式を採用する。つまり、共有バスを流れるすべてのメモリトランザクションを監視する機構が必要である。このメモリアクセスユニットとして調停式 MAU を採用する。この調停式 MAU の機能を以下に示す。

①優先度レジスタ

－メモリアクセス要求に優先度をつけ、調停を行う。

②ライトスルー方式

－キャッシュへの書き込みの度に共有メモリへも必ず書き込みを行う。

③書き込み更新

－キャッシュへの書き込みの際、他キャッシュに対しても書き込んだ値で更新する。

（2）大規模並列ハードウェアの例

対象とする問題が比較的大規模な場合、FPGA 内部 RAM の大きさなどを考慮に入れるため、複数 FPGA に複数のスレッドを実装する。それらに対して相互結合網を用いて、内部 RAM 及び外部 RAM と結合し共有分散メモリを構成する。このとき、一般的な相互結合網に広く用いられているディレクトリ方式を採用する。つまり、どのキャッシュがどのコヒーレンスブロックをキャッシングしているかという情報を共有メモリがディレクトリとして管理しておき、コヒーレンス維持の必要が生じた際に対象となるキャッシュに対してその旨を指示する手段をとる。このメモリアクセスユニットとして CC－NUMA 式 MAU を採用する。この CC－NUMA 式 MAU の機能を以下に示す。

① フルマップディレクトリ

－コヒーレンスブロックごとに、１プロセッサにつき１ビットを用いてどのプロセッサが当該プロセッサをキャッシングしているかをビットベクタの形で保持する。概念図を図１７に示す。

② ライトスルー方式

－キャッシュへの書き込みの度に共有メモリへも必ず書き込みを行う。

③ 書き込み更新

－キャッシュへの書き込みの際、他キャッシュに対しても書き込んだ値で更新する。

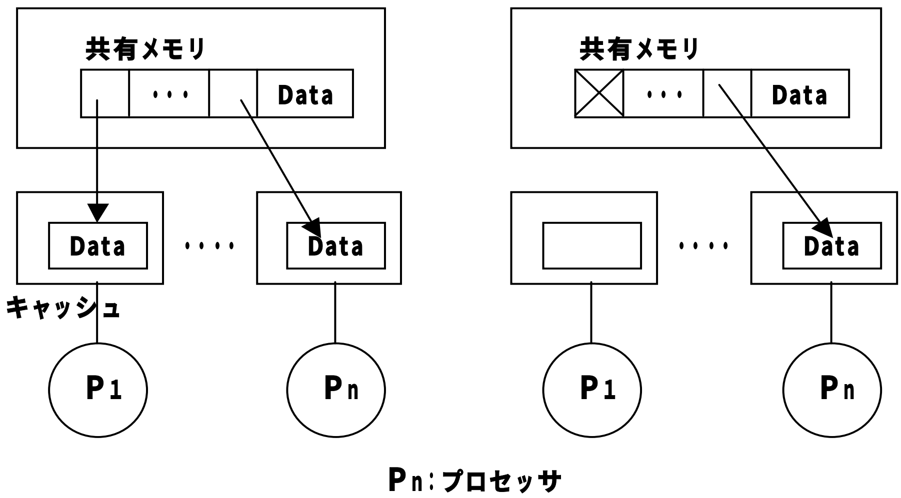


図 17　フルマップディレクトリ

同期処理は排他制御とバリア同期を考慮に入れる必要がある。排他制御に関しては、MAU での処理により複数スレッドが同時に共有メモリにアクセスできないことより、実現でき、バリア同期は３．５で述べた方法で実現できると考える。

4.3 並列ハードウェア構成例

４章でこれまで述べてきたハードウェア構成の例として、１FPGA 上に複数スレッドの場合を図１８に、１FPGA 上に１スレッドの場合を図１９に示す。

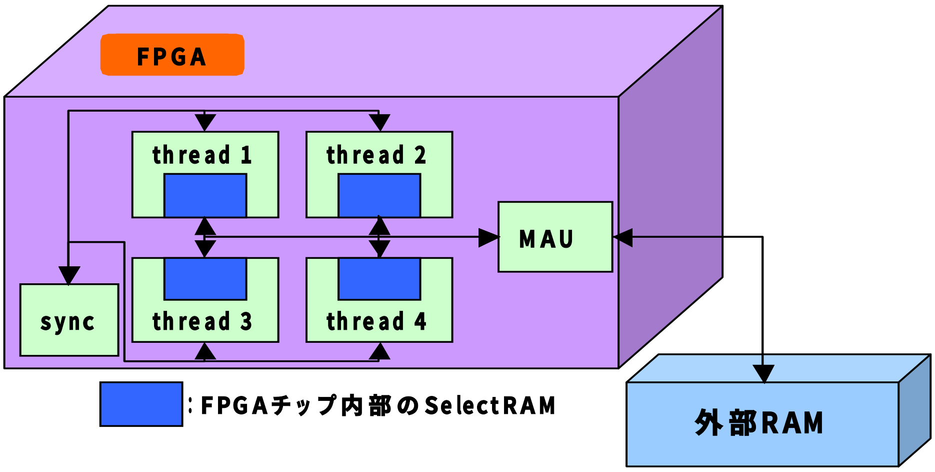


図 18　小規模並列ハードウェア構成例

この例では、１FPGA 上に４つのスレッドを実装し、FPGA チップに内蔵されている SelectRAM を用いて、キャッシュメモリを実現する。また、MAU は共有バスによって各スレッドと結合し、同期ユニットは、送信される同期信号をビットベクタの形で保持し、その and によりバリア同期を実現する。

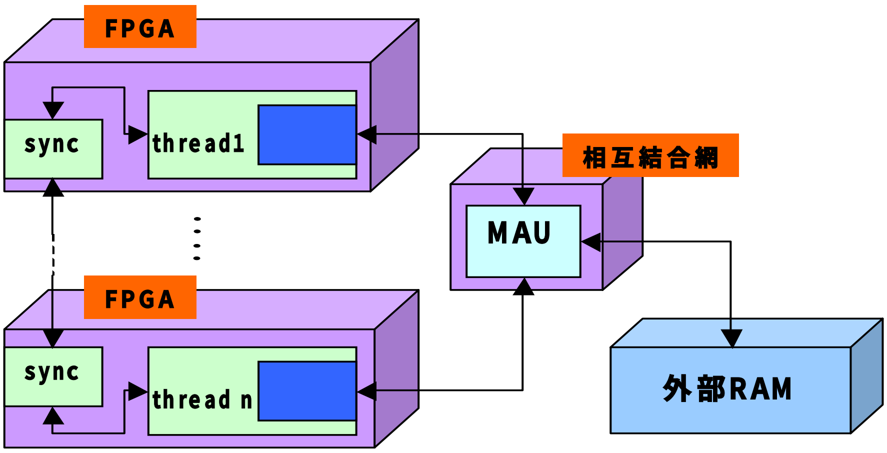


図 19　大規模並列ハードウェア構成例

この例では、１FPGA に１つのスレッドを実装し、SelectRAM を１スレッドのキャッシュメモリとして持つ。MAU は相互結合網により各スレッドと結合している。同期ユニットは、鎖状網により結合している。

5. まとめ

本研究では、共有メモリ型プログラミング言語である OpenMP をハードウェア記述言語である VHDL に変換することにより、その問題に対してコンパクトなハードウェアが生成できるシステムを検討してきた。システム構成においては、M コンパイラと T コンパイラの２つに分かれる。M コンパイラとは、３つのブロックに分かれており、OpenMP の並列構文を解析する並列構文解析部、ワークシェアリング構文及び変数のデータスコープ属性を処理する並列リージョン処理部、MAU 及び同期ユニットを構築する同期処理部について検討した。この M コンパイラによって、OpenMP が各スレッドに対する C プログラムとして変換される。T コンパイラは、この変換された C プログラムを VHDL に変換するコンパイラである。

また、並列実行を実行するためのメモリアクセスや同期に関する方式を問題の規模によって考察した。規模が小さい場合、すべてのスレッドを１FPGA 上に実装し、共有バスによって結合する。キャッシュコヒーレンスは、スヌープキャッシュ方式により回避し、メモリアクセスに優先度をつけることにより排他制御を実行する。規模が大きい場合、１つのスレッドを１つの FPGA 上に実装する。その際、各スレッドは相互結合網にて結合し、キャッシュコヒーレンスはディレクトリ方式にて回避する。さらに同期ユニットを鎖状網により結合することで、個別の FPGA（スレッド）間のバリア同期も実現する。

今後の課題として、このシステムの完成が挙げられる。本研究での考察を基に、OpenMP の構文を解析し、C プログラムに変換する M コンパイラを作成すれば、システムは完成する。また、複数の FPGA で実装する場合のハードウェア構成及び FPGA 間通信や同期処理に関する遅延を明確にし、より具体的な並列ハードウェアの実装に近づける必要がある。まずは、単純な OpenMP プログラムから FPGA 実装への段階を踏む事が大規模な並列ハードウェア自動生成を完成する足がかりになると思われる。

謝辞

本研究に関して貴重な助言、ご指導をいただきました山崎勝弘教授、西村俊和教授に感謝いたします。また、本研究に対して貴重なご意見をいただきました山内寛紀教授、山田喬彦教授、STARC 研究員の方々に感謝いたします。

また、ハードウェアの基礎知識や開発ツールの指導をしていただきました Tran Cong So 氏、OpenMP プログラミングに際して相談に応じてくれた岩田悠貴君、三浦誉大君、大村浩文君、及びいろいろな相談に応じてくださった本研究室の皆さんに感謝いたします。

参考文献

[1]佐藤茂久,草野和寛,佐藤三久:OpenMP 並列プログラムのデータフロー解析手法,JSPP2000,pp.71-76,2000.

[2]山中栄次,金子正教,堀田耕一郎:分散メモリ並列計算機向け OpenMP 拡張仕様,SHINING2001,HPC,pp.13-16,2001.

[3]佐谷野健二,片下敏弘,小池汎平.児玉裕悦,坂根広史,甲村康人:大容量 FPGA の応用によるマルチプロセッサエミュレーションシステムの評価,HOKKE2001,ARC,pp.25-30,2001.

[4]佐藤茂久, 草野和寛, 佐藤三久:OpenMP 向けコンパイラ支援ソフトウェア DSM の性能評価,SHINING2001,HPC,pp.7-12,2001.

[5]水谷泰治,中島大輔,藤本典幸,萩原兼一:並列再帰の実行方式をプログラマが指定可能なコンパイラの評価,電気情報通信学会論文誌 D-Ⅰ,Vol.J84-D-I,No.6,pp.594-604,2001.

[6]若林正樹,天野英晴:並列計算機シミュレータの構築支援環境, 電子情報通信学会論文誌 D-Ⅰ,Vol.J84-D-I,No.3,pp.247-256,2001.

[7]佐々木敬泰,西村直巳,弘中哲夫,吉田典可:マルチプロセッサ用スケジューリング支援ハードウェアの提案とシミュレーション評価, 電子情報通信学会論文誌 D-Ⅰ,Vol.J84-D-I,No.11,pp.1515-1531,2001.

[8]佐谷野健二,児玉裕悦,坂根広史,山口善教:MIPS ベースマルチスレッドプロセッサの FPGA による実装と評価,情報処理学会研究報告,2000-ARC-139,pp.151-156,2000.

[9]佐谷野健二,児玉裕悦,坂根広史,山口善教:並列計算機ノードプロセッサの FPGA を用いた実装と評価,情報処理学会研究報告,99-ARC-134,pp.49-53,2000.

[10]草野和寛,佐藤茂久,佐藤三久:Omni OpenMP コンパイラの性能評価,HOKKE2000,pp.179-184,2000.

[11]草野和寛,佐藤茂久,佐藤三久:Omni OpenMP コンパイラと実行時ライブラリの性能評価,JSPP2000,pp.229-236,2000.

[12]佐藤三久:Omni OpenMP コンパイラと Cluster-enabled OpenMP,京大大型計算機センター第６６回研究セミナー,2001.

[13]佐藤三久:OpenMP,新情報処理開発機構つくば研究センタ,2001.

[14]OpenMP C/C++ アプリケーションプログラム インタフェース バージョン 1.0-1998 年 10 月版:http://pbplab.trc.rwcp.or.jp/pbperf/Omni/spec.ja/omp-C-1.0/

[15]田中義久:C 言語からのハードウェア自動生成システムの構築,立命館大学大学院理工学研究科修士論文,2000.

[16]上平祥嗣:並列アルゴリズムクラスに基づくハードウェア自動生成,立命館大学大学院理工学研究科修士論文,2000.

[17]R.Chandra,L.Dagum,D.Kohr,D.Maydan,J.McDonald and R.Menon: Parallel Programming in OpenMP,MORGAN KAUFMANN PUBLISHERS,

[18]仁志彰宏:租粒度結合マルチプロセッサシステム自動生成法の研究,立命館大学大学院理工学研究科修士論文,1999.
[19]長谷川裕恭:VHDL によるハードウェア設計入門,CQ 出版社,1995.
[20]David A.Patterson,John L.Hennessy,成田光彰訳:コンピュータの構成と設計,第 2 版,上,日経 BP 社,1999.
[21]David A.Patterson,John L.Hennessy,成田光彰訳:コンピュータの構成と設計,第 2 版,下,日経 BP 社,1999.
[22]KITE マイクロプロセッサ・ボード PLUS＋取扱説明書 Version1.00,九州計測株式会社,1995.
[23]KITE マイクロプロセッサリファレンス・マニュアル Version1.00,九州計測株式会社,1993.
[24]KITE マイクロプロセッサ・クロスソフトウェア・ユーザーズ・マニュアル Version2.0,九州計測株式会社,1995.
[25]中田育夫:コンパイラの構成と最適化,朝倉書店,1999.
[26]Bil Lewis,Daniel J.Berg 共著/岩本信一訳: マルチスレッドプログラミング入門,アスキー出版,1996.
[27]佐藤 淳, 福田 孝一, 市田 真琴, 今井 正治, ”特定用途向きマイクロプロセッサ開発システムの構成方法に関する一考察”, 電子情報通信学会, 第 3 回 回路とシステム軽井沢ワークショップ 論文集, pp.74-81, 1990.
[28]今井 正治, ”ハードウェア／ソフトウェア・コデザイン -- 「ハードウェアの見積りと生成」”, 情報処理学会誌, Vol.36, No.7, pp.614-619, 1995.
[29]今井 正治, ”ハードウェア／ソフトウェア・コデザインへの期待と今後の課題”, 情報処理学会, DA シンポジウム '94 論文集, pp.217-222, 1994.
[30]今井 正治, ”Hardware/Software Codesign with Application Specific Processor”, ASP-DAC '95, Tutorial A, pp.73-93, 1995.
[31]安浦寛人, ”基本ソフトウェアとコデザイン”, 情報処理学会誌, Vol.36, No. 7, pp.620-625, 1995.
[32]G.de.Micheli, ”ハードウェア/ソフトウェア強調設計のコンピュータ支援における問題および方法について”, 情報処理学会誌, Vol.36, No. 7, pp. 605-613, 1995.
[33]末吉 敏則, 飯田 全広, “リコンフィギャラブル･コンピューティング”, 情報処理学会誌, Vol.40, No. 8,pp. 777-782, 1999.
[34]岩田 悠貴,田中 義久,山崎 勝弘:C 言語による有限ステートマシンベースのプロセッサ生成, 電子情報通信学会,VLD2001-117,pp.33-38,2001.
[35]松井 誠二,山崎 勝弘,”並列プログラムからのハードウェア自動生成システムの検討”,第 64 回情報処理学会全国大会,5E-04,2002.